O'REILLY®
Report

Code as Data

Governing and Evolving Large-Scale Codebases in the Era of AI

Bryan Friedman, Pat Johnson & Olga Kundzich





The agent tools company

moderne.ai

Code as Data

Governing and Evolving Large-Scale Codebases in the Era of AI

Bryan Friedman, Pat Johnson, and Olga Kundzich

O'REILLY®

Code as Data
by Bryan Friedman, Pat Johnson, and Olga Kundzich

Copyright © 2026 O'Reilly Media, Inc. All rights reserved.

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Louise Corrigan
Development Editor: Gary O'Brien
Production Editor: Jonathon Owen
Copyeditor: Paula L. Fleming
Cover Designer: Susan Thompson
Cover Illustrator: Ellie Volckhausen
Interior Designer: David Futato
Interior Illustrator: Kate Dullea

September 2026: First Edition

Revision History for the First Edition
2026-08-24: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Code as Data*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Moderne. See our statement of editorial independence.

979-8-341-67420-2

[LSI]

Table of Contents

1. Enterprise Software in the AI Era. 1
Impossible Questions About Your Codebase 2
AI Agents Don't Know Your Codebase 3
What This Book Is About 4

2. A New Operating Model for Software. 5
The Lossless Semantic Tree Code Model 5
Code as Data: What the LST Exposes 7
The LST at Work: From Data to Action 8
Knowledge Base Derived from the LST 11

3. Querying the Codebase—Finding the Unfindable. 15
Fast Search and Semantic Search 15
When the Searcher Is an Agent 16
Recipes as the Query Language 16
Indexed Search Built on the LST 17
Asking Your Own Questions 18
A Practical Search Workflow 19

4. Code Security and Advanced Program Analysis. 21
Tracing Data Through the Codebase 21
What Pattern Matching Misses 22
The Zero-Day Response 23
What Else Semantics Unlock That Inventory Cannot 25

5. Code Quality, Testing, and Dependencies. 27
Code Quality as Continuous Signal 27
Testing as a Continuous Safety Net 29
Dependency Hygiene Always 30
What Stops Being a Surprise 31

6. Code as Data Is the Foundation for Agentic Systems. 33
Real Context in Real Time 34
Act Once, Run Everywhere 35
Building Trusted Agentic Systems 36

CHAPTER 1
Enterprise Software in the AI Era

You have never seen your software. No one has. Not the engineer who architected it, not the team that owns the service today, and not the agent you just asked to make changes. Each of them works from an incomplete picture of your codebase that gets assembled fresh every time it's needed—and forgotten when it's not.

You notice it the moment you try to change anything. A routine security fix stalls because hundreds of applications are pinned to a version you have to upgrade first. A refactor that should touch one module reaches into dozens, because there's no record of what depends on it. A framework swap scoped for a weekend uncovers a decade of assumptions no one wrote down. The change itself is rarely the hard part. The hard part is that no one can see what all it touches until they're already inside it.

When developers were the ones making changes, one at a time, this was survivable. Now, however, agents are writing code at a volume no review process was built to absorb. Yet they still read it the same partial way developers always have, inferring through text what they can't see. You're being asked to trust change across the whole estate, faster than anyone can verify it.

This book is about what it would take to finally see your software. Model it once, with compiler accuracy, and let everyone (humans and agents) work from the same reference. This capability has never mattered more.

Impossible Questions About Your Codebase

Everything that developers and agents use to work with code—to read it, search it, reason about it—is a representation of the code, not the whole of it. And every one of these representations is inherently incomplete:

- The *IDE* resolves a developer's code, including types, symbols, and what calls what, but not beyond the project open on their machine.
- *Text representation of code* is stripped of executable meaning like types or symbols; it's storable and readable but lacks actionable precision.
- *Abstract syntax trees* describe syntax without fully resolving types and call targets across repositories.
- *Dependency inventories* show what is declared without revealing what is reachable or exercised.
- *Architecture catalogs* describe the system secondhand, providing an approximation that drifts as the system evolves.

The limits of today's representations are not theoretical. They add up to a code intelligence gap: the distance between what your software actually does and what your organization can see. This gap surfaces the moment you try to answer routine questions about how your systems are built and how changes ripple through them:

- Which services in your estate use a given library, and which of those actually invoke the vulnerable method rather than merely declaring the dependency?
- Which teams are still running Java 8 or .NET 6? Which of their services cannot be upgraded without first resolving transitive dependency constraints?
- Which services process user-supplied data, and where does that data cross trust boundaries inside your system?
- Where are the circular dependencies in your estate that would turn a simple upgrade into a cascade?
- Which methods in your codebase have no test coverage and high cyclomatic complexity, and which of those are called from dozens of places?



Questions like these are what platform, architecture, and security teams face every day. Use this list as a self-assessment. If your team cannot answer three or more of these questions within an hour, your organization has a code intelligence gap.

You might be tempted to assume that agents change this calculus. Ask one to trace a data flow or identify a vulnerable call site, and it will often move faster than a team of engineers. But it's working from the same incomplete code representations as everything and everyone else. The end result: agents reasoning in the dark—quickly, confidently, and sometimes getting it wrong.

AI Agents Don't Know Your Codebase

Coding agents are operators working directly inside the system. They plan tasks, navigate codebases, modify files, introduce dependencies, and open pull requests. And like every developer before them, they are hungry for context to complete their tasks.

Some connect to language server tooling or IDEs for symbol resolution within a single repository. Much of their work, however, still relies on searching code as text—reaching for `grep`, reading files to locate usages, unpacking library archives, and following import statements from one module to the next. Basically, they're building a working model of the code from the outside in.

What they can't know, they infer. This is a rational response to an incomplete information environment. But inference is inconsistent by nature: assembled differently each time, constrained by what the agent happened to discover, and disconnected from any authoritative representation of how the system actually fits together. It is also expensive to repeat at scale. Token cost accumulates as every agent reconstructs the same understanding from scratch across the same repositories again and again.

Agent-driven development is unrelenting, producing change faster than software ever has. A team's whole sprint becomes an afternoon for a fleet of agents. But agents run on probabilistic models, so they're nondeterministic, and getting it mostly right isn't good enough when a small error rate multiplies across thousands of decisions and hundreds of repositories. That's how bugs pile up faster than any review can catch them.

Slowing agents down isn't the answer. A deterministic foundation is. Agents need context and direction built on an accurate model of what the compiler knows about the code. Instead of having to inspect the code and infer behavior, the agent works from resolved data to produce results you can trust. That same foundation is how organizations understand the cumulative effect of thousands of decisions and intervene when the system drifts.

We're going to build remarkable things with agents—but only if they are working from the reality of the codebase, not an approximation. Chapter 2 details this deterministic foundation.

What This Book Is About

The chapters that follow cover the semantic models and analysis techniques that make it possible for humans and AI agents to reason accurately about large codebases and transform them reliably at scale. We've written this for engineering leaders, architects, and senior engineers responsible for large software estates who need to think clearly about what the AI era demands. The pace of change in AI means that some details will age quickly. However, the underlying shift to treating code as data will be the durable foundation for whatever comes next.

CHAPTER 2

A New Operating Model for Software

The impossible questions from Chapter 1 are impossible for a reason. Answering *Which services call the vulnerable method?* requires knowing more than the text representation reveals. You need to know the exact type behind each symbol, which library owns it, and the resolved call site behind every reference. Take one line: `logger.info("Payment received")`. SLF4J, Log4j, Logback, and Apache Commons Logging all expose a method named `info`. As text, they're indistinguishable. Without type attribution, a scanner flags every service that mentions the library and leaves developers and agents to sort out the real matches.

Treating the codebase as data—structured, semantic, scaled, and queryable—is the way forward, and in this chapter we'll introduce you to the code representation that makes it possible: the Lossless Semantic Tree (LST).

The Lossless Semantic Tree Code Model

The LST is built for understanding and working on code. It's the proven technology that underpins the open source OpenRewrite framework, as well as Moderne's platform and agent tools. It helps developers and coding agents understand and improve code more accurately and safely by preserving all the important details about how the code is organized and how its parts interact. The name is its specification:

Lossless, meaning nothing is discarded
Semantic, meaning types and symbols are fully resolved
Tree, meaning it's structured and traversable programmatically

Most static analysis tools resolve structure. Some may resolve symbols—binding names to their declarations within a project. The LST goes further: it achieves compiler-accurate type resolution. This means that it resolves types, generics, and dependencies across all of a project's libraries with the same fidelity the compiler brings when it builds the code. This applies for any language the LST supports, including Java, C#, Python, JavaScript, Scala, and others. This is the level at which meaning becomes unambiguous.



Think of the LST as a genome for your codebase. Every living thing has a genome: the complete, exact set of instructions that determines what it is and how it works. The LST carries the types, methods, and relationships that define how your code behaves. And just as one genome spans every cell in your body, the LST holds every signal the compiler sees across every repository, turning thousands of repos into one body of code.

Here's what this looks like in practice: every node in the LST carries not just its syntactic identity but its full semantic context, as shown in Figure 2-1. A method call node knows which library's method it refers to, including the fully qualified type that owns it, the resolved return type in this specific calling context, and the resolved types of every argument passed to it. A class declaration node knows every interface it implements, its full supertype chain, and every member it declares.

Formatting, such as whitespace, comments, and style, is also preserved in the LST alongside every node, so when a recipe modifies the tree and writes it back to source, the output is idiomatic with the original source.

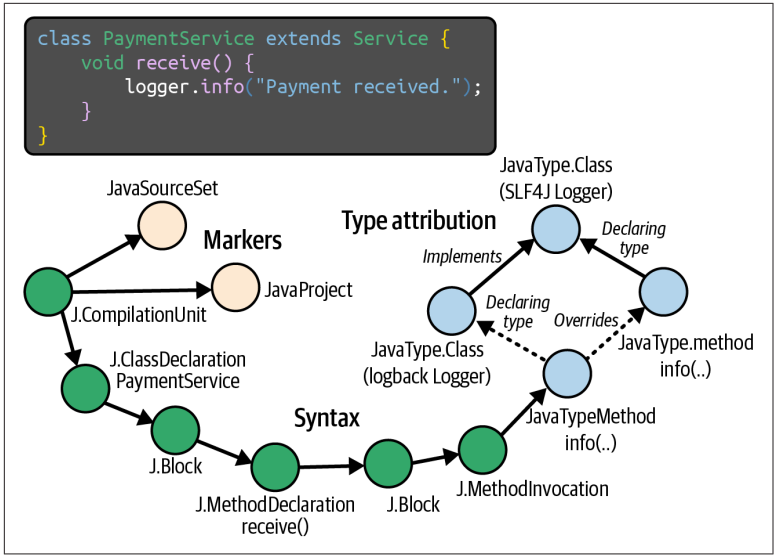


Figure 2-1. The LST's three layers—markers for annotation, syntax for structure, and type attribution for fully resolved meaning—illustrated with a simple Java `logger.info` call

Code as Data: What the LST Exposes

The codebase contains at least seven categories of structured data. Each answers different questions, but in practice they overlap and reinforce each other. Let's take a tour of what becomes answerable once code representation is compiler-accurate:

Semantic knowledge
: Resolved types, symbols, call graphs, and method signatures across files and projects. This is the foundation that everything else is built on; without it, you have text, not data.

Data flow knowledge
: How values move through the system: control flow, taint analysis, and reachability from entry point to exit. This is the backbone of security analysis—what makes it possible to answer whether a vulnerability is actually reachable, not just declared.

Quality knowledge
: Complexity, coupling, cohesion, test coverage/gaps, and hotspot concentration at the method level. This is exportable as structured data that agents and analytics tools can consume directly.

Dependency knowledge
: Not just which libraries are declared, but which are actually used, at what version, and with what transitive exposure. This is the foundation of dependency hygiene and open source risk management.

Architectural knowledge
: The real dependency graph, actual layering, and coupling between modules—including structural dependencies invisible to single-repo tools. This is the prerequisite for governing change at scale.

Compliance knowledge
: Real-time, code-grounded evidence that regulatory requirements and organizational policies demand. This includes software bills of materials (SBOMs), license obligations, and known vulnerability inventory.

Infrastructure knowledge
: Build files, CI/CD pipelines, and infrastructure as code (IaC) definitions that are treated as first-class code artifacts, queryable across the estate. A misconfigured identity and access management (IAM) role in a Terraform file carries as much risk as a vulnerability in application code.

The LST at Work: From Data to Action

The LST is built for two realities of modern software: enterprise scale and agent-driven development. Understanding how it works explains why it is fit for both. Figure 2-2 provides an overview of putting the LST to work, from building it to using it.

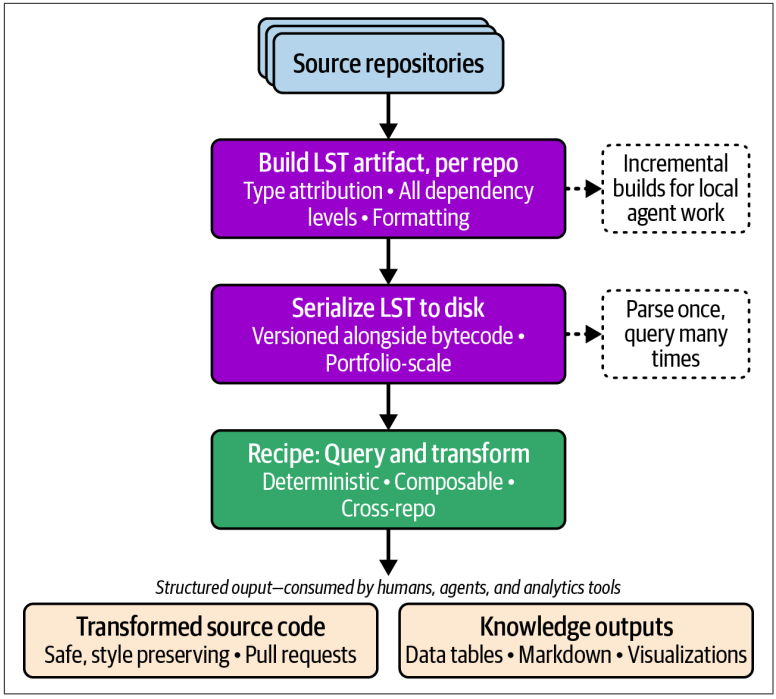


Figure 2-2. Putting LSTs to work for your codebase: from build to knowledge and transformation

Building LST Artifacts

An *LST artifact* reflects the full contents of a repository. It is produced by guiding the compiler through the first two phases of compilation, syntactic and semantic analysis, to produce full type attribution. The result is a complete model of the code with every type resolved, every dependency traversed, and every formatting detail preserved. LST files can be built in bulk as well as incrementally to support real-time coding.

Incremental builds enable only what changed to be reparsed and re-resolved, without invalidating the rest of the stored model. A commit that modifies three files in one service triggers re-resolution of those files and the affected portions of the semantic graph, while the rest of the portfolio remains available for querying immediately. Crucially, the LST supports random access of files during the build itself: the portions of the model already built are available for search, analysis, and transformation before the full build completes.

Serializing the LST for Portfolio-Scale Work

LSTs are built per repository, but they become far more useful when they are serialized to disk. Stored as a by-product of the normal compilation process, they are versioned and retrievable in your artifact repository. This transforms LSTs from ephemeral build artifacts into persistent, queryable assets. It's the difference between a database that exists only while a query runs and one that persists between queries.

Consider a shared internal library with an API change. Which consuming services, across dozens of repositories owned by different teams, are affected, and which call sites need to change? No single repository's LST can answer that, but the serialized portfolio of LSTs can. Parse once, query many times—this is what makes portfolio-scale analysis fast.

The Recipe: LST Query and Transformation Language

A *recipe* is a reusable program that traverses the LST, visits nodes matching a pattern, and either marks them (in the case of a search) or transforms them (in the case of an update or remediation). Unlike ad hoc scripts or probabilistic AI suggestions, recipes are deterministic: the same input always produces the same output. Recipes can run across multiple LSTs at once, searching, analyzing, and transforming code simultaneously across repositories.

Recipes may be simple: find every usage of a deprecated method, bump a dependency version, or enforce a naming convention across a repository. Here is one example of a recipe that renames a deprecated method:

```
type: specs.openrewrite.org/v1beta/recipe
name: com.yourorg.RenameGetItems
displayName: Rename Catalog.getItems() to listItems()
recipeList:
  - org.openrewrite.java.ChangeMethodName:
      methodPattern: com.yourorg.Catalog getItems()
      newMethodName: listItems
```

The `methodPattern` only matches `getItems()` on the `Catalog` class:

```
catalog.getItems()  →  catalog.listItems()
cart.getItems()  →  cart.getItems()  (untouched, different type)
```

A find-and-replace would rename both. Since the recipe knows the types, it touches only the calls that actually resolve to `Catalog`.

This declarative format also allows recipes to be composable so they can be combined into complex workflows. A framework migration involves thousands of coordinated changes expressed as a sequence of smaller, testable transformations rather than a single monolithic script. For example, a Spring Boot 4.x migration chains together recipes that update dependencies, rename packages, replace deprecated APIs, and adjust configuration files.

In addition, recipes can emit data tables, which are structured outputs produced as a by-product of traversing LSTs. A recipe searching for vulnerable dependencies, for example, simultaneously produces a table of every affected call site, its severity, the minimum fixed version, and its location across the portfolio. These tables are exportable in standard formats and consumable directly by analytics tools, dashboards, and AI agents.

And you don't have to write recipes from scratch. The OpenRewrite community offers a large and growing catalog, with vendors providing technology-specific recipes. These are backed by extensive unit tests that run each recipe against a broad set of real-world before-and-after examples. Organizations can customize or build their own recipes, increasingly using AI agents as authors. Once an agent has explored a codebase, it can capture a repeatable change as a recipe that runs deterministically across the portfolio, instead of editing thousands of files one by one.

Knowledge Base Derived from the LST

To see what code as data looks like in practice, consider Prethink: a composite recipe that runs across repositories, extracts what the LST knows, and produces a knowledge base of structured files that are usable by humans and agents alike.

Prethink produces outputs across several dimensions:

- A complete dependency inventory—direct and transitive—across all repositories
- A knowledge graph that maps how components, services, and behaviors connect across the estate, surfaced as CALM-formatted architecture files

- Code quality metrics—including complexity, coupling, cohesion, and package-level circular imports—that pinpoint where the code is most fragile
- Test gap analysis that ranks untested methods by risk score, so engineers and agents know where code is most likely to break when it changes
- Code duplication data that identifies exact, renamed, and gapped clones, as well as near duplicates that exact matching misses

For example, Prethink computes complexity scores for every method in the codebase and writes the results to a data table. The method risk radar in Figure 2-3 plots the worst offenders from the table, one polygon per method, across five dimensions normalized to a common scale: cognitive complexity, cyclomatic complexity, estimated bugs, parameter count, and nesting depth.

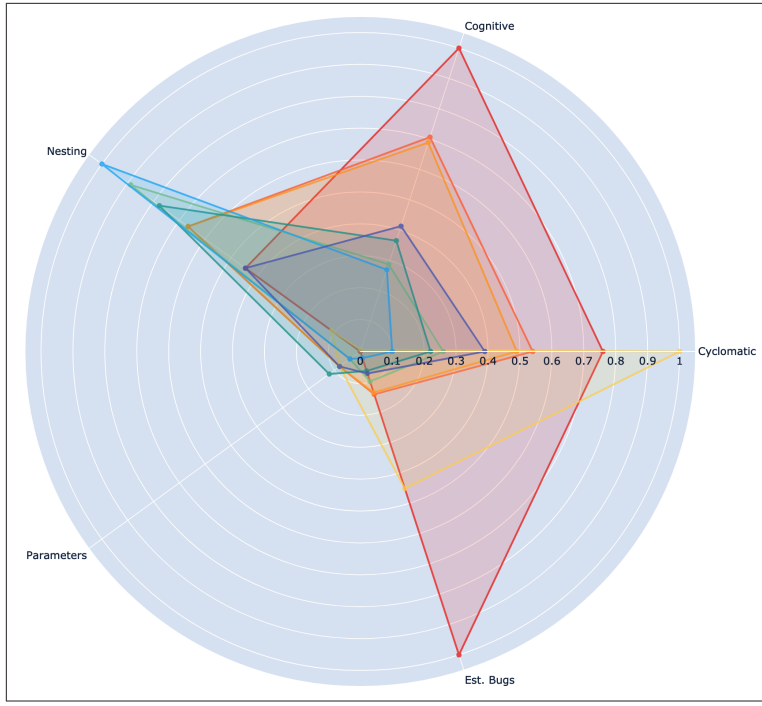


Figure 2-3. Radar chart, where each polygon represents one of the riskiest methods in the codebase, profiled across five dimensions

Methods that reach furthest from the center carry risk on several axes at once, and an agent knows to treat them as fragile. Before touching one, it reads the method closely, keeps new logic out, and weighs splitting it rather than adding branches to code that is already near its limit.

Prethink materializes all the data as versioned Markdown and comma-separated value (CSV) files, committed to the repository and referenced by the agent's config file (e.g., *CLAUDE.md* or *.cursorrules*). An agent opening the repo reads the files automatically and knows which methods are complex, which classes to split, which packages are in cycles, and where the riskiest test gaps are. Rerun Prethink after any change, and every agent picks up the update without spending tokens rediscovering what changed.

This is what "code as data" looks like in practice: not a dashboard that humans triage but structured, authoritative knowledge derived from the code itself. The chapters that follow show you how to use code as data—how to query it, secure it, and put it in the hands of the agents increasingly doing the work.

CHAPTER 3

Querying the Codebase—Finding the Unfindable

Every nontrivial change to a codebase starts with these questions: *Where is this used? Who depends on this? Which services still call this deprecated API?* The answers determine what gets read, what gets edited, and what eventually ships. For most of software history, the tool answering these questions was some flavor of text search with the same operation underneath: *Where does this string appear?* That worked when codebases were small and the questions were simple. Most large codebases now are knowable in theory and unknowable in practice. Even though the information is there, the tools used to find it can see only the surface of the code, not what it actually means.

Chapter 2 introduced the Lossless Semantic Tree as the representation that makes a codebase queryable as structured data. This chapter is about how that capability actually works in practice.

Fast Search and Semantic Search

Not every question requires the same level of precision. Text search tools like `grep` or "find in files" are fast, ubiquitous, and useful when false positives don't matter. Use them to find a distinctive configuration key, a specific string literal, or an identifier unique enough that a certain amount of noise is acceptable. Abstract syntax tree (AST)–based tools go a step further, matching the structural

shape of code rather than just strings, but they still stop short of resolving what types and symbols actually refer to.

The problem is that text and structural search reach their limits when the questions being asked can't be answered without a deeper understanding of types, dependencies, call graphs, and resolved imports. After all, two methods named `getUser()` may not be the same method. `Logger` from Log4j and `Logger` from SLF4J share a name but not a type.

Semantic search reasons at the same level as a compiler, so every identifier is resolved to what it actually refers to. IDEs solve this within a single project, but LST-based semantic search unlocks this capability across repository boundaries.

When the Searcher Is an Agent

Exploration is something a coding agent does constantly. Every time an agent reads a codebase, it's exploring—searching for a string, opening files to see what they contain, and searching again based on what it has found. The economics of that loop are very different from a developer's. When an agent greps and gets a noisy result, every candidate file gets opened and read in full. That read loop after the search can be particularly expensive. Every result, including false positives, consumes tokens and fills the context window with material the agent doesn't actually need.

The fix is to provide a search whose results are meaningful enough that the read loop is unnecessary: if the result already says "method declaration named `process`, in `com.example.PaymentService`, public, returns `void`," there's no need to open anything.

Agent tools backed by the LST enable more efficient, accurate agents. Deterministic tools like index-based search for fast, symbol-aware exploration and semantic search recipes for precise, type-resolved answers keep the agent focused on the task.

Recipes as the Query Language

As with any data model, getting answers requires a query language. In a traditional database, you write SQL. For a codebase treated as data, you write a recipe (introduced in Chapter 2). Recipes operate directly on the LST, with access to everything the compiler resolved

during the build phase—types, symbols, transitive dependencies, and all.

Consider a `FindMethods` recipe. It locates every call to a specific method across hundreds of repositories, identified by the method's fully qualified type rather than its name—a distinction that text search can't make. Run `grep -r "StringUtils"` across a set of Java repositories, and it returns every reference of any kind: imports, calls, comments. There's no way to tell Apache Commons' `StringUtils` from Spring's, or a live call from a mention in a comment. `FindMethods` disambiguates the type, so it returns only the calls that invoke the method you actually asked for. Here's the query phrased as a recipe:

```
mod run . --recipe=org.openrewrite.java.search.FindTypes \
-PfullyQualifiedTypeName="org.apache.commons.lang3.StringUtils"
```

This returns only the actual usages of Apache Commons' `StringUtils`, ignoring the other `StringUtils` classes that happen to share a name across libraries.

Indexed Search Built on the LST

Whether the searcher is a developer or an agent, recipe-level precision is overkill when the goal is simply to find a string or to explore an unfamiliar codebase where a fast, repeatable query loop matters more than a complete one. That kind of search has a well-established solution: the *trigram index*, which finds a string across thousands of repositories in milliseconds. Pioneered by Russ Cox at Google, it now underlies systems like GitHub code search, Sourcegraph, and Zoekt—all of which index raw source. It scales, but it inherits the false positives that come with matching text.

Moderne Trigrep starts from that same trigram foundation but makes the queries smarter. It uses a symbol-aware index backed by the LST's full type table, so it carries complete symbol data: the kind of each identifier, the methods a class declares, and the supertypes it implements. The familiar trigram query language is then extended with predicates that draw on that symbol data. The result is the symbol awareness of semantic search at the speed of a text index, allowing you to query based on what an identifier is, not just what it's named, across thousands of repositories at once. For an agent,

that pairing of speed and accuracy shortens the discovery loop by providing precise matches the first time:

```
# Type-aware predicate: every class that extends RestOperations
mod search . type:class extends:RestOperations

# Composition: every public method returning ResponseEntity
mod search . visibility:public type:method \
 returns:ResponseEntity
```

These queries ask questions about the code that no text-only index can answer with confidence. Note that indexed search over the LST can still return occasional false positives, since the index isn't a full traversal, but it is still more semantically aware than a basic text search. That makes it more useful when you're exploring the code. It's also helpful when you're making a first pass, where the next step will be to use a recipe that can resolve the remaining ambiguity (or transform the code) precisely.

Asking Your Own Questions

One of the most practically valuable things you can do with a semantic query system is to take an honest inventory of what your code actually does. In large organizations, stated inventories often diverge from real ones:

- A migration that was supposed to standardize on PostgreSQL reveals 15 services still talking to the old Oracle instance.
- A framework consolidation uncovers three different teams that independently chose different HTTP client libraries.
- A Java version upgrade finds repositories still running on Java 8 that no one knew existed.

A recipe that answers the question *What Java version is each repository actually using?* produces ground truth in a data table: one row per repository with actual version information extracted directly from the code. The same approach surfaces any standards drift that has occurred, such as repositories using more than one logging framework, services bypassing the approved connection pool, or teams lagging on a framework adoption.

Recipes can answer questions like these because they're programs, not configurations. A tool that ships with a fixed set of queries can only answer the questions its authors anticipated. A recipe can

express any query that is stateable in terms of the LST, including queries specific to your domain. A company using LaunchDarkly, for instance, wrote a recipe that called LaunchDarkly's API to retrieve unused flags, used that data to drive code transformations to remove the associated conditional branches, and then deleted the unused flags from LaunchDarkly itself. No general-purpose tool could have done this. It was possible because recipes are programs with access to external APIs, conditional logic, and multiphase execution, and so can turn what would otherwise be an occasional manual cleanup into a standing operation.

A Practical Search Workflow

Putting all the pieces together, a typical pass through a real problem (e.g., a deprecated API, a vulnerable dependency, or an architectural review) looks something like this:

1. *Scope the portfolio with an indexed search.* Which repositories in the portfolio even reference the API in question? At portfolio scale, this eliminates most of the work before any expensive analysis runs.
2. *Qualify the working set with a semantic predicate.* Among the candidates, which usages match the patterns that need attention? Predicates like `extends:`, `returns:`, and `visibility:` are most helpful here.
3. *Resolve to define the target with a recipe.* A deterministic OpenRewrite recipe walks the LST across the candidates and produces either precise, type-resolved answers to a search query or a completed transformation ready to ship.

Figure 3-1 visualizes this workflow.

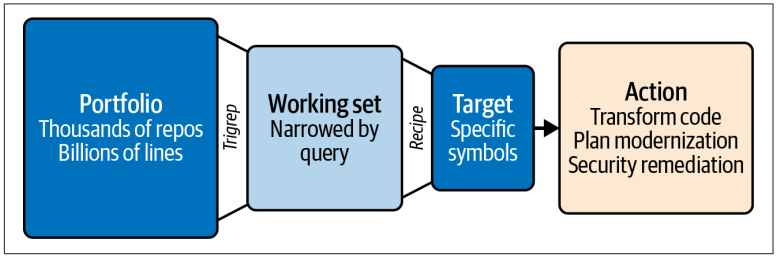


Figure 3-1. Workflow from search to action, where each stage shrinks the scope

The structure is the same no matter whether a developer or an agent is orchestrating it. Each step reduces uncertainty and concentrates the work where it matters. Migrations can be planned against an accurate account instead of an estimation. Vulnerabilities can be qualified by whether the affected path is actually reachable. Architectural reviews can be conducted against the dependencies the code actually has.

Search has traditionally been asked to answer these questions. But with the LST underneath, search can now run at whatever level the question requires, from a fast text scan to a precise, type-resolved analysis and across one repository or the whole portfolio.

Chapter 4 takes the same semantic foundation a layer further, into the program analyses that turn structured queries into operational understanding of control flow, data flow, and how the code actually behaves.

CHAPTER 4

Code Security and Advanced Program Analysis

Software composition analysis (SCA) tools inventory known vulnerable dependencies. Static application security testing (SAST) tools flag vulnerabilities in first-party code. And now frontier AI models are surfacing zero-day vulnerabilities not yet described in vulnerability databases. The volume of findings keeps rising. But these tools mostly answer the question *What do you have?* Advanced program analysis answers the harder questions, enabling you to identify vulnerabilities accurately by tracing calls across systems.

That distinction matters everywhere software change happens (whether by human or agent), including vulnerability remediation, framework migrations, dead code elimination, and architectural governance. These problems require reasoning about software behavior, execution paths, data flow, and relationships across entire portfolios. Built on the LST and composable in deterministic recipes, advanced program analysis turns code as data into something operational: a system for producing accurate, actionable answers.

Tracing Data Through the Codebase

Tracing calls across systems rests on two long-standing techniques. *Control flow analysis* maps how a program can execute, such as which statements are reachable, which branches are dead, or which exceptions can escape. *Data flow analysis* walks through that map and tracks where values come from and where they end up. What

lets it scale is method summarization: it computes once what each method does to the data passing through it and then reuses that summary everywhere the method is called. That's what turns an analysis that works on one file into one that works across hundreds of millions of lines of code.

In practice, an analysis like this is packaged within a recipe and follows the call graph the LST already encodes. *Taint analysis* is data flow analysis aimed at a security question: you state once where untrusted data enters (a source) and where it could do harm (a sink), and the recipe follows the value between them, flagging any path that reaches a sink without passing through a sanitizer first. The pattern generalizes to any risk defined by what reaches what: injection, data leakage, credential exposure, or access control gaps.

The recipe works because of the code model underneath it. The analysis has to resolve each call to the specific method it actually invokes, not guess the method from a matching name. It has to tell `user.name` (from a request) apart from `user.id` (generated server-side), or it floods you with false alarms. And because real systems span many services, it has to follow data as it leaves one repository and enters another. That precision is possible only because the LST resolves symbols with compiler accuracy and holds that fidelity across every repo.

What Pattern Matching Misses

CVE-2026-22732 is a Spring Security header suppression vulnerability in which a controller commits the response too early by flushing the output stream, writing past a buffer threshold, or setting `Content-Length` explicitly. This causes Spring Security's lazy header filter to silently drop critical security headers.

A pattern-matching rule keyed on `response.getOutputStream().write(...)` catches this case. But the same vulnerability shows up in shapes that the rule does not match (see Figure 4-1). Each is the *same* bug at the data flow level—a value from one API reaches another API where it commits the response—but each looks different on the surface.

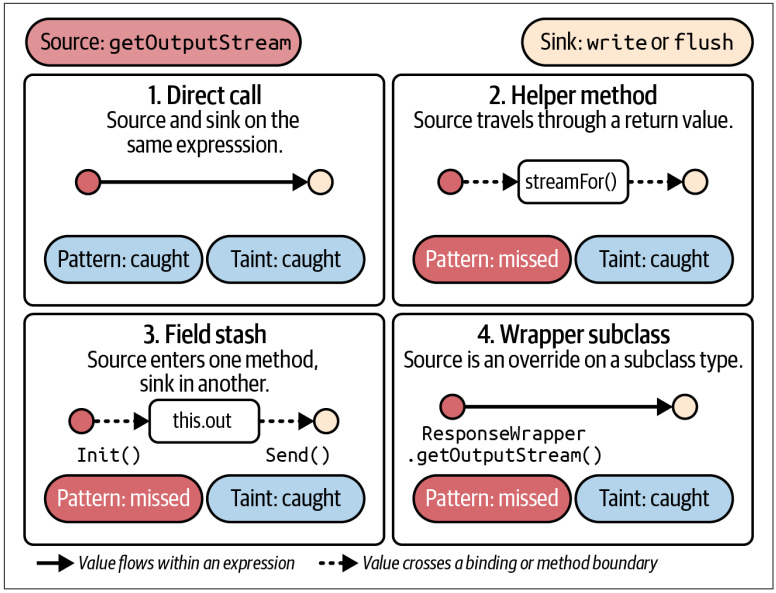


Figure 4-1. Identifying a vulnerability with multiple shapes: pattern matching versus taint analysis

Taint analysis catches all of them because it tracks the value, not the syntax. The source ("the result of calling `getOutputStream` on a `ServletResponse` or any subclass") and the sink ("a `write` or `flush` on the receiver of that value") are stated once. The analysis handles the rest: following the stream across helper methods, through field stores and loads, and through wrapper subclasses and propagating constant strings across local bindings.

The Zero-Day Response

AI is reshaping the vulnerability landscape on multiple sides. Frontier models like Anthropic's Mythos are being used to find zero-day vulnerabilities in first-party code and in the third-party dependencies embedded throughout modern software estates. Meanwhile, developers and their agents are producing more code (and introducing new vulnerabilities) faster than reviewers can keep up. Wherever a defect comes from and however it's identified, the next questions are the same: *How exposed are we? Which findings actually matter? Where are they reachable? Can we eliminate them everywhere they exist?*

The response still needs to be fast: it needs to identify the vulnerability at scale and distribute the fix across every repository that needs it, all in one motion. This is what recipes are built for. When the Spring Security vulnerability appeared, one enterprise running 3,000 services needed to know whether the vulnerability could be remediated across every one of them quickly, consistently, and with certainty. The business needed a deterministic yes, not a probabilistic probably.

While the advisory describes Spring Security's server-side behavior, it can't tell you which of your 3,000 Spring repositories actually contain code paths that trigger it, because that depends on what your code does. We used agentic coding to research the vulnerability and build the recipe. The striking aspect of this is how little the agent had to build. Almost all of the code is already there to pull from, such as taint flow, field sensitivity, type-aware matching, and the propagation and receiver checks. All this is analysis that's already running in production recipes.

Constructing the detection was mostly composition, pulling together one line naming the source, one naming the sink, a data-table binding, and a wiring to the upgrade dependency version recipe that applies the fix. This demonstrates the compounding property of a recipe library built on a shared analysis foundation. It's why this solution came together in an afternoon and why the next vulnerability of the same shape will be cheaper still to remediate. Once built, it runs across the estate, detecting the affected repositories and upgrading only those that require the patch. Figure 4-2 shows the find-and-fix process at scale.

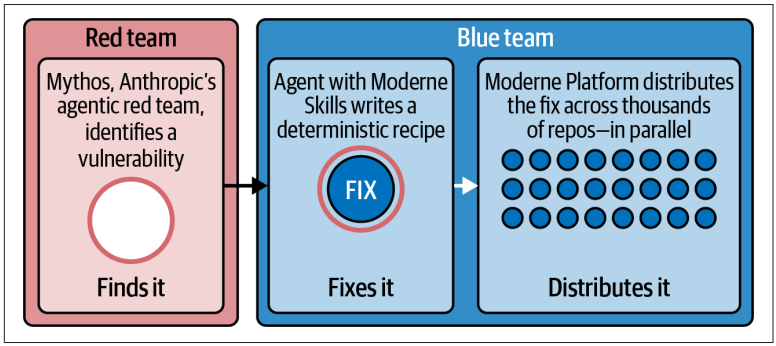


Figure 4-2. Example of a zero-day response at enterprise scale

Why Not Just Ask the Agent to Fix It?

When agentic AI takes on advanced program analysis, the output is probabilistic: it might be right or wrong, complete or partial. The model can't resolve types without guessing, follow every branch through a closure, or reliably find and fix issues across a codebase. The common pattern is that the agent identifies the issue, fixes a few occurrences, declares it done, and leaves the rest behind.

Therefore, it's better to have the agent produce a deterministic recipe you can test, reuse, and improve than to ask the agent to improvise each attempt. Recipes run faster, more consistently, and far more cheaply than sending the codebase through a frontier model every time the same question comes up.

What Else Semantics Unlock That Inventory Cannot

The same techniques (LST + recipes) that address security concerns like CVE-2026-22732 can also answer questions across the codebase that inventory-based tools like SCA cannot. And in every case, the findings come back precisely located, typed, and connected to the exact code constructs that produced them, actionable by developers and agents:

Post-quantum cryptography (PQC) readiness
: Control flow and data flow analysis can follow how values like hardcoded keys, insecure defaults, and outdated algorithm choices propagate through function calls and class hierarchies and across repository boundaries to find quantum-vulnerable encryption.

Safe refactoring and dead code
: Whether it's a variable that's never read, a method no entry point can reach, or a parameter that's never used, it's a data flow or call graph question. Answering it reliably is what makes a "remove unused method" recipe safe to run across the portfolio.

Migration impact analysis

When an API is deprecated, the actionable question is not *Who imports it?* but *Who can actually reach it from a live code path, and with what arguments?* Migrations that look 10 times bigger than they are often shrink under semantic analysis.

Architectural conformance

Does the presentation layer ever call the data layer directly, despite what the diagram says? Data flow across the call graph answers that empirically rather than aspirationally.

Chapter 5 uses the same semantic foundation to answer a different question: *What kind of quality, testing, and dependency data keep large codebases coherent over time?*

CHAPTER 5

Code Quality, Testing, and Dependencies

Coding agents produce and change code faster than anyone can review it. They'll confidently modify a method with a cyclomatic complexity of 47, zero test coverage, and 200 call sites depending on it, where a careful engineer would just leave it alone. And an agent not only runs into code like this, but it also generates it. The same data on quality that steers an agent away from risky code catches the risky code it just wrote.

Quality dashboards, coverage reports, and dependency scans all live on a server for a human to review. To an agent that opens a repo and acts in seconds, information that isn't clearly accessible is, functionally, information that doesn't exist (or must be inferred). This chapter is about three categories of codebase knowledge—quality, tests, and dependencies—and why agents and humans both need them as real-time data, resolved from the code itself and current at the moment of the next change.

Code Quality as Continuous Signal

Code quality used to be a question of whether the code was readable, testable, and understandable by the next engineer. That still holds. But agents add a more urgent question: *What is the health of the codebase at the moment the next change starts?* A 78% maintainability score on a dashboard doesn't help an agent that just opened a 2000-line class.

Complete quality data needs to answer questions at three altitudes, because problems at one level are usually symptoms at another:

Method level
: Signals like cyclomatic and cognitive complexity, nesting depth, and Halstead-derived estimated bugs tell the next actor how risky a single method is to modify.

Class level
: Cohesion and coupling—measures like Lack of Cohesion in Methods (LCOM4)—produce directly actionable signals. "LCOM4 = 3" doesn't say a class has problems; it says it should be split into three. That's a different output than a score.

Package level
: Dependency cycles, abstractness, and instability surface architectural decay that no method-level view can see. For example, Moderne analyzed 100+ enterprise repos and found that nearly half of all packages were caught in dependency cycles. The quality tooling already in place was not surfacing this.

Composite thresholds turn metric combinations into named code smells like God Class, Feature Envy, and Data Class. Each comes with the metric evidence behind it. "Move this method because it touches more fields on another class than its own" is a meaningfully different signal than a complexity score.

All of those signals—method-level risk, class-level structure, named smells—roll up into a composite technical debt score per method. It's visualized as a treemap sized by code volume and colored by that score, as shown in Figure 5-1. The biggest, reddest blocks are the highest-return refactoring targets.

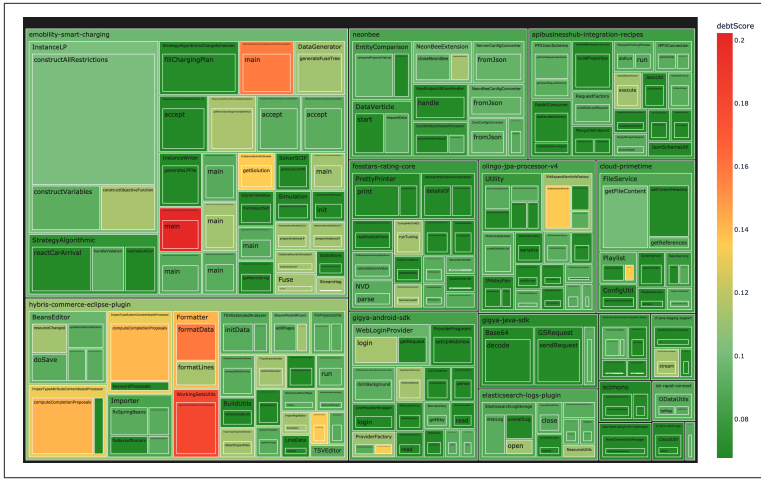


Figure 5-1. Technical debt treemap identifying high-value refactoring targets

This is the version an engineering leader can read in five seconds without knowing what *cyclomatic complexity* means. Point at the big red block, say "Fix that one first," and you've prioritized your portfolio. The agent reads the same data in real time to decide whether to refactor before adding new behavior.

Testing as a Continuous Safety Net

Even before agents, test coverage was only part of the testing story. Now with agents in the loop, it's wholly inadequate. A repo at 78% coverage can have dozens of untested high-complexity, high-traffic methods, and the percentage doesn't tell you which ones they are. Therefore, an agent given that number won't know it should write tests before modifying the code.

The signal that matters is the one that indicates which untested methods are dangerous to touch. Test gap analysis (as shown in the Figure 5-2 heatmap) inverts coverage. Every untested method gets a risk score combining complexity and call count. A complex method called from 40 places scores in the high-risk hundreds; an untested getter scores in trivial single digits.

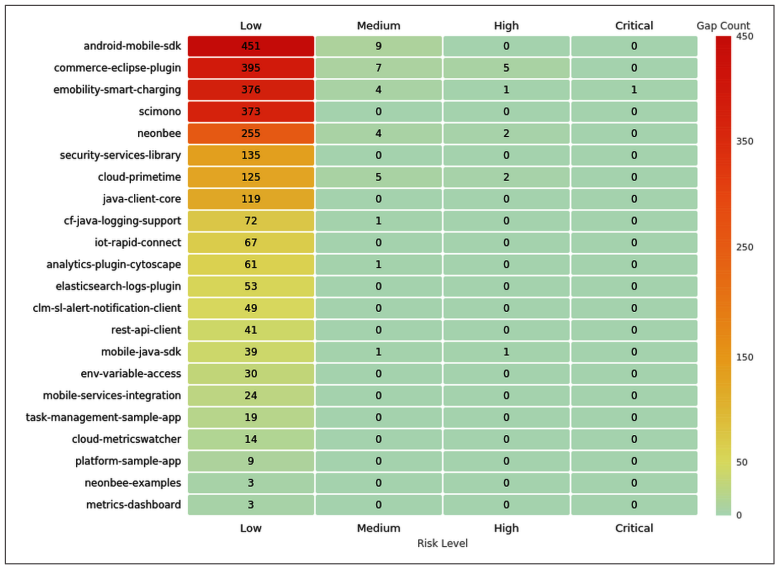


Figure 5-2. Test gap heatmap—untested methods bucketed by risk score across repos

The gap list surfaces only the methods that actually warrant tests. In a 129-repository portfolio analysis, filtering left 2,765 untested methods that genuinely warranted attention. For a head of engineering, this visualizes a situation where one wrong edit by a developer in a hurry or an agent in a loop can take production down.

When the dangerous gaps are sitting in the repo as ranked data, however, the sequence of work changes: write the test on the highest-risk untested method first and then make the change. Each entry even includes a suggested test class, giving agents a starting point rather than a blank page. Tests stop being an artifact of finished work and become a precondition for safe change—for agent and developer alike.

Dependency Hygiene Always

The bulk of any modern codebase is third-party code, and the picture of what's actually there changes constantly. New transitive dependencies arrive with every upgrade, vulnerabilities get disclosed, and versions drift. That's a problem for agents.

An agent reasoning about a change that touches a library—such as adding a call, refactoring an existing one, or evaluating impact—needs the current, complete picture. It needs every direct and transitive dependency, the versions in play, and where each library is actually used across the codebase. When it is inferred from raw text files, that picture is incomplete and often wrong. Pulled from a scanner's last run, it's stale. The only version of this data that's useful is the one resolved from the code itself and current as of right now.

For dependencies, "current, resolved state in the repo" means four things:

The resolved dependency graph
: Direct and transitive, with versions, depth, scope, and license. So the question *Are we exposed to this CVE—and how deeply?* is a lookup rather than a full investigation.

Where each dependency is used
: Which types from each library are referenced, where, and how often. This requires resolved types, not text search; a `grep` for `ObjectMapper` would miss `JsonMapper` and `XmlMapper`.

Known vulnerabilities mapped to the resolved versions present
: Not "This library has a CVE" but "This version of this transitive dependency at depth 4 is vulnerable, and here's the minimum fix."

A narrative of how the codebase uses its libraries
: For the developer reviewing a pull request and the agent reasoning about a change.

When the dependency picture is current and resolved, keeping the dependencies themselves current becomes a tractable problem too. Before upgrading to a new version of a library, an agent can compare the new version to the current one and see what changed. The same resolved data that tells an agent "This transitive dependency is vulnerable" tells an upgrade recipe what to bump. Knowing and changing operate on the same data model.

What Stops Being a Surprise

Quality, tests, and dependencies are three of the most expensive recurring problems in enterprise software. They show up as the refactor that was supposed to take a day and instead takes a week,

the production incident traced back to a method nobody had tested, and the Friday afternoon vulnerability in a transitive dependency four levels deep. And the compounding cost underneath all of it is that engineering teams are busier than ever but delivering less of what the business actually needs.

That changes when your teams and agents can rely on the codebase itself as the source of truth. The moment an agent opens the code, real-time data for quality, tests, and dependencies is right there. This is what solutions like Moderne Prethink provide. None of the problems disappear: third-party libraries will keep changing, code will keep accumulating complexity, and tests will keep falling behind. But the shape of the work changes. The surprises stop being surprises. And with coding agents, limiting surprises is the whole game.

Chapter 6 picks up at this point, making the case that code as data isn't just useful for agents but is the foundation they need to operate.

CHAPTER 6

Code as Data Is the Foundation for Agentic Systems

Coding agents are now part of how software gets built. More than that, they've triggered the most significant transformation the software lifecycle has seen in decades. The debate about the next generation of language models continues, but practically, the conversation has focused on agent orchestration—agent selection, coordination strategy, subagents, plan mode, role-scoped workflows, MCP servers, and elaborate harnesses. However, the real conversation starts further down, at the source.

Agentic AI runs on context at scale. An agent's output is only as good as what it can see, including not only the file in front of it but also dependencies, callers, and conventions across the whole codebase. When that signal is missing, the LLM infers and guesses. In a large enterprise codebase, the consequences compound, creating bottlenecked review cycles, verbose and needlessly complex code, confidently incomplete modernizations, and ratcheting token costs.

That's why the LST, the code model we introduced in Chapter 2, becomes a valuable layer between the agent and the source. It holds types, dependencies, and call graphs, all resolved and queryable across thousands of repositories as a single corpus. Figure 6-1 shows how the LST fits between your codebase and the agent activity. The rest of this chapter explains what that layer makes possible.

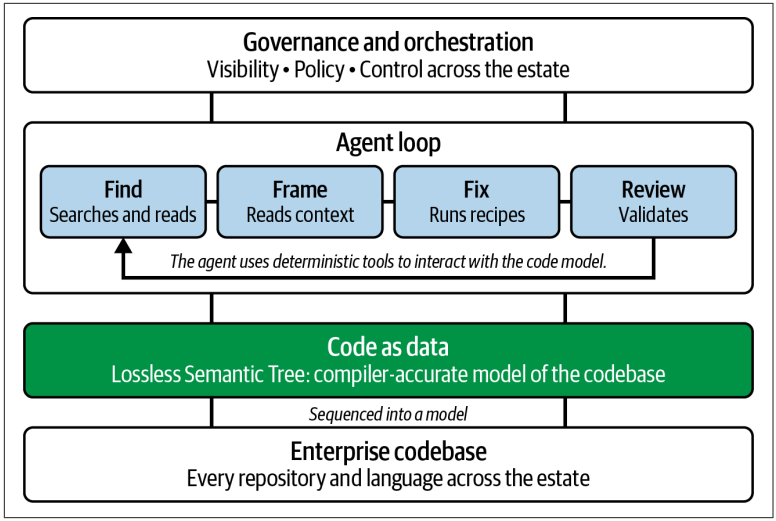


Figure 6-1. Enabling agents to reason against the code-as-data layer, not the raw source as text

Real Context in Real Time

An agent's first move on any task is discovery, because it must find what it's working with before it can change anything. Against raw text, that's a wide-ranging hunt, reading file after file toward an understanding that it must rebuild from scratch next time. Using the LST instead, agents discover with precision so they reach real, actionable code understanding faster.

As we saw in Chapter 3, an agent's exploratory loop favors fast, repeatable queries. The difference here is what the responses provide. When the index is built from the LST, even a search that returns in milliseconds is able to distinguish a method's actual callers from those that just share its name. The agent gets text-search speed with accurate type information underneath, so it can spend its reasoning on making the change, not on finding what to change.

A precomputed knowledge base (highlighted in Chapter 5) frames the work up front, sparing the agent from reinvestigating the codebase each session. This saves tokens and sharpens the agent's judgment. After all, the more an agent knows, the better the code quality will be. Take a God Class, for example. This is code that is doing far too much, with its responsibilities all tangled together, making it difficult to understand, test, and evolve. An agent reading a few

files at a time has a hard time recognizing this risky structure and will cheerfully add one more method, making a bad case worse. However, when the agent is given quality data up front, it already knows the class is overloaded. It can cite the evidence and give new logic its own class, instead of piling on.

Likewise, handed test gap data, an agent can see which paths are unguarded and write the missing tests first, changing code behavior on safer ground than if editing blind. Handed full dependency resolution, the agent can reason about the version on the runtime path three layers down, not just the one in the import line. That can be the difference between a fix that holds and one that compiles but then breaks in production.

The context already exists in the code. Tools like Moderne's Trigrep search and its Prethink knowledge base deliver the context as data that is available the moment the agent needs it. The agent's guesses turn into informed judgments and use just a fraction of the tokens.

Act Once, Run Everywhere

Understanding code and safely changing it at scale are two different problems. For example, a migration to Spring Boot 4.x is really dozens of coordinated actions:

- The version bump cascades into Spring Framework, Hibernate, and Jackson.
- Config keys get renamed deep in YAML.
- Monolithic starters split into modular ones according to actual usage.
- Build files change across Maven and Gradle.
- Deprecated APIs must be removed or rewritten.

This is why an agent turned loose with file access and a build tool often makes real progress but then stalls. It fixes the obvious imports but misses the config keys it hasn't even seen. Then it burns millions of tokens compiling to discover what broke and gets partway to the answer in a handful of repositories. Worse, the agent actually thinks it's finished. Multiply this result across thousands of repositories that also need the upgrade, and the math collapses.

A recipe carries the key to the migration that the agent lacks: steps that are independently tested and correctly sequenced. With a recipe, the resulting code passes verification and compiles. The agent doesn't have to try to reason through each change. Instead, it orchestrates the recipe run, which applies the same changes consistently to one repository or a thousand with minimal token overhead. A recipe can stamp out a migration on 80% to 90% of applications, leaving agents to apply their reasoning only to the edge cases.

Code review looks different too. You sign off on the recipe once—validating its logic and that it ran identically everywhere—not on ten thousand diffs. Determinism turns thousands of approvals into one. Human judgment goes where it's actually needed: the model-generated changes, where no test can vouch for the result. As agents write more code than anyone can review, recipes give back critical time.

NOTE

Recipes don't just pay off once. An agent that solves a novel problem can capture the solution as a new recipe, which is tested and repeatable. What one agent figures out becomes a deterministic capability that every future agent inherits. The system gets more capable the more it's worked.

Building Trusted Agentic Systems

Across this book, we've turned code from text you read into data you can query, analyze, and change. That's the foundation that engineering teams build on, and for agents, it's the difference between guessing and knowing.

Running on real code data builds trust in agentic systems. When agents reason against resolved facts, change code through tested recipes, and leave a record you can audit, they are predictable and more cost-effective. You don't have to check every change to know it will hold. The firmer the ground underneath them, the further fleets and factories of agents can go before a human needs to step in, because the work is verifiable.

Going further at this scale takes more than the code. It takes a record of the change around it: agent transcripts, change history across the organization, and who changed what and why. That

record is data too, queryable alongside the code. It's what lets agents reason soundly about how a large codebase is evolving and lets humans supervise at the levels that matter.

We've shown what code as data makes possible today. Now imagine what will get built on this foundation as agentic systems grow more capable. The teams that treat their code as data will be the ones who find out.

About the Authors

Bryan Friedman is director of product management at Moderne, where he works on automated code transformation and agent tooling built on the open source OpenRewrite project. He has spent his career on software platforms and developer enablement, first in enterprise IT and application development, and then in developer relations and technical marketing, working with engineering and IT teams on how large organizations upgrade, migrate, and maintain code at scale. He is a frequent writer and speaker on code modernization and developer enablement.

Pat Johnson has spent her career bringing technically complex products to market for developers and operators. She's director of product marketing at Moderne, which gives AI coding agents a deterministic foundation to write, refactor, and modernize code accurately at enterprise scale. Previously, she led portfolio marketing for VMware's Tanzu platform and drove product marketing for CI/CD solutions at Pivotal and CA Technologies. Earlier in her career, she was a technical writer and marketer in the networking semiconductor space.

Olga Kundzich builds enterprise software that stands up at scale. She is cofounder and CTO at Moderne, where she's delivering the deterministic foundation that lets coding agents operate efficiently and cost-effectively across enterprises with billions of lines of code. She has charted Moderne's product and technology direction from the start, keeping it aligned with the real needs of enterprise customers. Previously, she was a technical product manager focused on application delivery and management at Pivotal and a lead software engineer and manager at Dell EMC, where she worked closely with enterprise users to put data protection practices into operation.